Int	::=	the domain of (unbounded) integer numbers, with usual operations on them
Bool	::=	the domain of booleans
VarId	::=	standard identifiers
AExp	::=	*Int* | *VarId* | *AExp* `+` *AExp* | *AExp* `/` *AExp*
BExp	::=	*Bool* | *AExp* `<=` *AExp* | `not` *BExp* | *BExp* `and` *BExp*
Stmt	::=	`skip` | *VarId* `:=` *AExp* | *Stmt* `;` *Stmt* |
		`if` *BExp* `then` *Stmt* `else` *Stmt* | `while` *BExp* `do` *Stmt*
Pgm	::=	`vars` **List**`{`*VarId*`}` `;` *Stmt*

Figure 3.1: Syntax of IMP, a small imperative language, using algebraic BNF

3.1 IMP: A Simple Imperative Language

To illustrate the various operational semantics styles, we have chosen a small imperative language, called IMP. The IMP language has arithmetic expressions which include the domain of arbitrarily large integer numbers, boolean expressions, assignment statements, conditional statements, while loop statements, and sequential composition of statements. All variables used in an IMP program are expected be declared at the beginning of the program, can only hold integer values (for simplicity, there are no boolean type variables in IMP), and are instantiated with default value 0.

3.1.1 IMP Syntax

We here define the syntax of IMP, first using the Backus-Naur form (BNF) notation for context-free grammars and then using the alternative and completely equivalent mixfix algebraic notation (see Section 2.5). The latter is in general more appropriate for semantic developments of a language.

IMP Syntax as a Context-Free Grammar

The syntax of IMP using the algebraic BNF notation is depicted in Figure 3.1. The only "algebraic" feature is the use of **List**`{`*VarId*`}` for variable declarations (last production), which in this case is clear: one can declare a comma-separated list of variables. To stay more conventional in notation, we refrained from replacing the productions "*Stmt* ::= `skip` | *Stmt* `;` *Stmt*" by the algebraic production "*Stmt* ::= $\mathbf{List}^{\texttt{skip}}_{\texttt{;}}$`{`*Stmt*`}`" which captures the idea of statement sequentialization more naturally; indeed, the former yields ambiguous parsing (luckily, the semantics of statement sequential composition will be such that the parsing ambiguity is irrelevant, but that is not always the case).

The IMP language constructs have their usual imperative meaning. For diversity and demonstration purposes, when giving the various semantics of IMP we will assume that `+` is *non-deterministic* (it evaluates the two subexpressions in any order, possibly interleaving their corresponding evaluation steps), `/` is non-deterministic and *partial* (it will stuck the program when a division by zero takes place), `<=` is *left-right sequential* (it first evaluates the left subexpression and then the right subexpression), and that `and` is left-right sequential and *short-circuited* (it first evaluates the left subexpression and then it conditionally evaluates the right only if the left evaluated to true).

To expose some of the limitations of the existing operational approaches, in Section 3.8 we extend IMP with expression side effects (an increment operation on variables), with abrupt termination (a

sorts:

$Int,\ Bool,\ VarId,\ AExp,\ BExp,\ Stmt,\ Pgm$

subsorts:

$$Int,\ VarId < AExp$$
$$Bool < BExp$$

operations:

$$\_\texttt{+}\_ : AExp \times AExp \to AExp$$
$$\_\texttt{/}\_ : AExp \times AExp \to AExp$$
$$\_\texttt{<=}\_ : AExp \times AExp \to BExp$$
$$\_\texttt{and}\_ : BExp \times BExp \to BExp$$
$$\texttt{not}\_ : BExp \to BExp$$
$$\texttt{skip} : \to Stmt$$
$$\_\texttt{:=}\_ : VarId \times AExp \to Stmt$$
$$\_\texttt{;}\_ : Stmt \times Stmt \to Stmt$$
$$\texttt{if}\_\texttt{then}\_\texttt{else}\_ : BExp \times Stmt \times Stmt \to Stmt$$
$$\texttt{while}\_\texttt{do}\_ : BExp \times Stmt \to Stmt$$
$$\texttt{vars}\_\texttt{;}\_ : \mathbf{List}\{VarId\} \times Stmt \to Pgm$$

Figure 3.2: Syntax of IMP as an algebraic signature

halt statement), and with dynamic threads. The extension of IMP with side effects, in particular, makes the various evaluation strategies of `+`, `<=` and `and` semantically relevant.

We will assume available the domains of integers and booleans, as well as basic operations on them which are clearly tagged (e.g., "$+_{Int}$" for addition of integer numbers) to distinguish them from homonymous operations which are IMP language constructs.

We may tacitly use the following naming conventions for terms or variables throughout the remainder of this chapter: $x, X \in Var$; $a, A \in AExp$; $b, B \in BExp$; $s, S \in Stmt$; $i, I \in Int$; $t, T \in Bool$; $p, P \in Pgm$. Any of these can be primed or indexed.

IMP Syntax as an Algebraic Signature

Following the relationship between the CFG and the mixfix algebraic notations explained in Section 2.5, the BNF syntax in Figure 3.1 can be associated the entirely equivalent algebraic signature in Figure 3.2 with one (mixfix) operation per production: the terminals mixed with underscores form the name of the operation and the non-terminals give its arity. This signature is easy to define in any rewrite engine or theorem prover; moreover, it can also be defined as a data-type or corresponding structure in any programming language. We next show how it can be defined in Maude.

☆ Definition of IMP Syntax in Maude

Using the Maude notation for algebraic signatures, the algebraic signature in Figure 3.2 can yield the Maude syntax module in Figure 3.3. We have additionally picked some appropriate precedences and formatting attributes for the various language syntactic constructs.

The module `IMP-SYNTAX` in Figure 3.3 imports three "builtin" modules, namely: `INT`, which we assume it provides a sort `Int`; `BOOL`, which we assume provides a sort `Bool`; and `VAR` which

```
mod IMP-SYNTAX is including INT + BOOL + VAR .
--- AExp
  sort AExp .  subsorts Int VarId < AExp .
  op _+_ : AExp AExp -> AExp [prec 33 gather (E e) format (d b o d)] .
  op _/_ : AExp AExp -> AExp [prec 31 gather (E e) format (d b o d)] .
--- BExp
  sort BExp .  subsort Bool < BExp .
  op _<=_ : AExp AExp -> BExp  [prec 37 format (d b o d)] .
  op not_ : BExp -> BExp [prec 53 format (b o d)] .
  op _and_ : BExp BExp -> BExp [prec 55 format (d b o d)] .
--- Stmt
  sort Stmt .
  op skip : -> Stmt [format (b o)] .
  op _:=_ : VarId AExp -> Stmt [prec 40 format (d b o d)] .
  op _;_ : Stmt Stmt -> Stmt [prec 60 gather (e E) format (d b noi d)] .
  op if_then_else_ : BExp Stmt Stmt -> Stmt [prec 59 format (b o bni n++i bn--i n++i --)] .
  op while_do_ : BExp Stmt -> Stmt [prec 59 format (b o d n++i --)] .
--- Pgm
  sort Pgm .
  op vars_;_ : List{VarId} Stmt -> Pgm [prec 70 format (nb o d ni d)] .
endm
```

Figure 3.3: IMP syntax as an algebraic signature in Maude. This definition assumes appropriate modules `INT`, `BOOL` and `VAR` defining corresponding sorts `Int`, `Bool`, and `VarId`, respectively.

we assume provides a sort `VarId`. We do not give the precise definitions of these modules here, particularly because one may have many different ways to do it. In our examples from here on in the rest of the chapter we assume that `INT` contains all the integer numbers as constants of sort `Int`, that `BOOL` contains the constants `true` and `false` of sort `Bool`, and that `VAR` contains all the letters in the alphabet as constants of sort `Var`. Also, we assume that the module `INT` comes equipped with as many "builtin" operations on integers as needed.

To avoid operator name conflicts caused by Maude's operator overloading capabilities, we urge the reader *not* to use the Maude builtin `INT` and `BOOL` modules, but instead to overwrite them. Appendix A.1 shows one possible way to do this in Maude 2.4: we define new modules `INT` and `BOOL` "hooked" to the builtin integer and boolean values but defining only a subset of operations on them and with names clearly tagged as discussed above, e.g., `_+Int_ : Int Int -> Int`, etc.

Recall from Sections 2.4 and 2.8 that lists, sets, bags and maps are trivial algebraic structures which can be easily defined in Maude; consequently, we take the freedom to use them without definition whenever needed, as we did with using the sort `List{VarId}` in Figure 3.3.

To test the syntax, one can now parse various IMP programs, such as:

```
Maude> parse
     vars n, s ;
     n := 100 ;
     s := 0 ;
     while not(n <= 0) do (
       s := s + n ;
       n := n + -1
     )
.
```

Now it is a good time to define a module, say `IMP-PROGRAMS`, containing as many IMP programs

```
mod IMP-PROGRAMS is including IMP-SYNTAX .
  ops sumPgm collatzPgm countPrimesPgm : -> Pgm .
  ops collatzStmt multiplicationStmt primalityStmt : -> Stmt .
  eq sumPgm = (
       vars n, s ;
       n := 100 ;  s := 0 ;
       while not(n <= 0) do (
         s := s + n ; n := n + -1
       )     ) .
  eq collatzStmt = (
       while not (n <= 1) do (
         s := s + 1 ; q := n / 2 ; r := q + q + 1 ;
         if r <= n
           then n := n + n + n + 1
           else n := q
       )     ) .
  eq collatzPgm = (
     vars m, n, q, r, s ;
     m := 10 ; s := 0 ;
     while not (m <= 2) do (
       n := m ; m := m + -1 ;
       collatzStmt
     )     ) .
  eq multiplicationStmt = ( --- fast multiplication (base 2) algorithm
    z := 0 ;
    while not(x <= 0) do (
      q := x / 2 ; r := q + q + 1 ;
      if r <= x then z := z + y                    --- if x % 2 == 1
      else skip ;
      x := q ;
      y := y + y
    )     ) .
  eq primalityStmt = (
    i := 2 ; q := n / i ; t := 1 ;
    while (i <= q and 1 <= t) do (
      x := i ; y := q ;
      multiplicationStmt ;
      if n <= z then t := 0 else (
        i := i + 1 ; q := n / i
      )
    )     ) .
  eq countPrimesPgm = (
     vars i, m, n, q, r, s, t, x, y, z ;
     m := 10 ; s := 0 ; n := 2 ;
     while n <= m do (
       primalityStmt ;
       if 1  <= t then s := s + 1 else skip ;
       n := n + 1
     )     ) .
endm
```

Figure 3.4: IMP programs defined in a Maude module `IMP-PROGRAMS`

as one bears to write. Figure 3.4 shows such a module containing several IMP programs. Note that we took advantage of Maude's rewriting capabilities to save space and reuse some of the defined fragments of programs as "macros". Defining such a module helps us to test the desired language syntax (Maude will report errors if the programs that appear in the right-hand-sides of the equations are not parsable), and will also help us later on to test the various semantics that we will define.

3.1.2 IMP State

Any operational semantics of IMP needs some appropriate notion of *state*, which is expected to map program variables to integer values. Moreover, since IMP disallows uses of undeclared variables, it suffices for the state of a given program to only map the declared variables to values and stay undefined and disallow updates of variables which were not declared.

Fortunately, all these desired IMP state operations correspond to conventional mathematical operations on *partial finite-domain functions* from variables to integers in $[\mathit{VarId} \rightharpoonup \mathit{Int}]^{\mathit{finite}}$ (see Section 2.1.2) or, equivalently, to equationally defined structures of sort $\mathbf{Map}\{\mathit{VarId} \mapsto \mathit{Int}\}$ (see Section 2.3.2 for details on the notation and the equivalence); we let *State* be an alias for the map sort above. From a rewriting logic semantic point of view, the equations defining such map structure are invisible: semantic transitions that are part of various IMP semantics will be performed *modulo* these equations. In other words, state lookup and update operations will not count as computational steps, so they will not interfere with or undesirably modify the intended computational granularity of the defined language (IMP in this case).

We let σ, σ', σ_1, etc., range over states. By defining IMP states as partial finite-domain functions $\sigma : \mathit{Var} \rightharpoonup \mathit{Int}$, we have a very natural notion of undefinedness for a variable that has not been declared and thus initialized in a state: state σ is considered *undefined* in a variable x if and only if $x \notin \mathit{Dom}(\sigma)$. We may use the terminology *state lookup* for the operation $\_(\_) : \mathit{State} \times \mathit{VarId} \to \mathit{Int}$, the terminology *state update* for the operation $\_[\_/\_] : \mathit{State} \times \mathit{Int} \times \mathit{VarId} \to \mathit{State}$, and the terminology *state initialization* for the operation $\_ \mapsto \_ : \mathbf{List}\{\mathit{VarId}\} \to \mathit{Int}$; recall from Sections 2.1.2 and 2.3.2 that the latter operation builds a partial function mapping each element in the first list argument (these elements are expected to be distinct) to the element given as second argument.

☆ Definition of IMP State in Maude

Figure 3.5 adapts the generic Maude definition of partial finite-domain functions in Figure 2.2 for our purpose here: the generic sorts `A` (for the source) and `B` (for the target) are replaced by `VarId` and `Int`, respectively, and the definition of state update is simplified to only update variables that are already in the domain of the map (this suffices for IMP). The only reason for which we bother to give this obvious module is because we want the various subsequent semantics of the IMP language, all of them including the module `STATE` in Figure 3.5, to be self-contained and executable in Maude by simply executing all the Maude code in the figures in this chapter.

```
mod STATE is including INT + VAR .
  sort State .
  op _|->_ : List{VarId} Int -> State [prec 0] .
  op .State : -> State .
  op _,_ : State State -> State [assoc comm id: .State format(d s s d)] .
  op _(_) : State VarId -> [Int] [prec 0] .        --- lookup
  op _[_/_] : State Int VarId -> State [prec 0] .  --- update

  var Sigma : State .  var X X' : VarId .  var Xl : List{VarId} .  var I I' : Int .

  eq (Sigma, X |-> I)(X) = I .
  eq (Sigma, X |-> I)[I' / X ] = (Sigma, X |-> I') .
  eq (X,X',Xl) |-> I = X |-> I, X' |-> I, Xl |-> I .
  eq .List{VarId} |-> I = .State .
endm
```

Figure 3.5: The IMP state defined in Maude